Trustworthy Data from Untrusted Databases

Rohit Jain, Sunil Prabhakar
Department of Computer Sciences, Purdue University
West Lafayette, IN, USA
{jain29,sunil}@cs.purdue.edu

***Abstract*—Ensuring the trustworthiness of data retrieved from a database is of utmost importance to users. The correctness of data stored in a database is defined by the faithful execution of only valid (authorized) transactions. In this paper we address the question of whether it is necessary to trust a database server in order to trust the data retrieved from it. The lack of trust arises naturally if the database server is owned by a third party, as in the case of cloud computing. It also arises if the server may have been compromised, or there is a malicious insider. In particular, we reduce the level of trust necessary in order to establish the authenticity and integrity of data at an untrusted server. Earlier work on this problem is limited to situations where there are no updates to the database, or all updates are authorized and vetted by a central trusted entity. This is an unreasonable assumption for a truly dynamic database, as would be expected in many business applications, where multiple clients can update data without having to check with a central server that approves of their changes.**

We identify the problem of ensuring trustworthiness of data at an untrusted server in the presence of transactional updates that run directly on the database, and develop the first solutions to this problem. Our solutions also provide indemnity for an honest server and assured provenance for all updates to the data. We implement our solution in a prototype system built on top of Oracle with no modifications to the database internals. We also provide an empirical evaluation of the proposed solutions and establish their feasibility.

I. Introduction

Ensuring the integrity and authenticity of data is of ever increasing importance as data are generated by multiple sources, often outside the direct control of fully trusted entities. Increasingly, data are subjected to environments which can result in invalid (malicious or inadvertent) modifications to the data. Such possibilities clearly arise when we host our data in a cloud computing setting where we lack complete control over the hardware and software running at the cloud servers. They can also arise when the data are maintained on trusted servers, but the data may get modified by a malicious insider or an intruder that manages to compromise the server or the communication channels. In these situations, *can we be ensured that the data retrieved from an untrusted server are trustworthy (i.e., the data and retrieved values have not been tampered or incorrectly modified)?*

With the advent of cloud computing there is increasing interest to move operations, including databases, onto a cloud platform. Although cloud computing holds great promise, it raises a number of security and privacy concerns. It also raises concerns about the fidelity of the service. In particular, since the clients have little or no direct control over the software and hardware that is running at the servers, there is reluctance to blindly trust the server. For example, a server may be improperly configured, or inadvertently left open to hacker attacks. There is also the concern about the server being attacked by an external entity that can corrupt the outsourced data or service despite the best efforts of the server. Even though, the cloud service provider is likely to be honest, it may try to hide its failure. Currently, users have no recourse but to trust the server or rely on legal agreements. Even with such agreements, it is difficult for a user to discover, let alone prove, any foul play by the server.

In this paper we show how it is possible to force an untrusted (relational) database server to provide trustworthy data. This is achieved by engaging the server in a protocol that makes it impossible for it to hide unfaithful execution. A key challenge for this work arises from the fact that multiple, independent clients can access and make valid updates to parts of the data. In order to ensure authenticity it is necessary to guarantee:

- **Correctness**: All answers to a query do indeed come from the authentic database.
- **Completeness**: The query answer contains all relevant tuples (i.e., no part of the answer is dropped).
- **Transactional Integrity**: The database always reflects a valid consistent state – i.e., the state corresponding to the initial state followed by the correct application of all previous valid transactions in the correct order. Furthermore, each new transaction executes against the latest (or freshest) state.

Of these, only the first two have been studied in earlier work. In most earlier work it was assumed that the data were not modified at the untrusted server. In other words, all updates were authenticated by the data owner and then sent to the database server. The legitimacy of any data that was part of the database was established directly by the data owner. In a dynamic database setting this is an unacceptable assumption. A typical database supports a large number of authorized clients that run transactions directly on the database. Updates to the data are made through these transactions. It is infeasible for the database owner to determine the correct updates for each transaction. The validity of these updates (i.e, what items are modified, and their new values) is determined by the faithful execution of a transaction semantics over the latest valid state. *How can the database owner be assured that the (untrusted) server is indeed correctly executing all transactions?*

Many applications may also require, e.g., due to regulatory compulsions, to track the provenance of updates to the database. This can be particularly important to check if malicious activity occurred in the past. In addition to these requirements from the data owner's perspective, there is an additional requirement from the service provider. The server should be able to prove its innocence if it has faithfully executed all transactions.

To the best of our knowledge, this problem of ensuring transactional integrity over an untrusted database server, as critical as it is for outsourced databases, has not been addressed in earlier work. The contributions of this work are:

- Identification of the problem of ensuring *Transactional Integrity* for databases hosted on untrusted servers.
- *Novel authentication mechanisms* that ensure correctness, completeness, and transactional integrity.
- Solutions that provide *indemnity for the server*, and also *trustworthy provenance* for the database.
- A demonstration of the feasibility of the solution through a *prototype in Oracle*, and its evaluation.

The rest of this paper is organized as follows. Section II presents some preliminary tools that are necessary for this work and summarizes existing results that form the basis of our solution. Section III presents our protocols. A discussion of the implementation of the solution and empirical evaluation is presented in Section IV. Section V discusses related work, and Section VI concludes the paper.

II. Preliminaries

In this section we present some basic tools that we use for building our solutions, and also discuss existing solutions for ensuring completeness and correctness.

A. Tools

We use three data security tools in this paper: *strong one-way hash functions*, *Merkle Trees*, and *digital signatures*.

1) One-Way Hashing: A one-way hash function h takes as input a data item x and produces the hash of the data item $y = h(x)$. Important requirements for a one-way hash function are: i) Given a hash value y, and the hash function h, it is infeasible to find x such that $h(x) = y$; and ii) It is infeasible to find two different data items, x and y, such that $h(x) = h(y)$. Table I summarizes the symbols used in this paper.

2) Merkle Hash Trees: A Merkle Hash Tree (MHT), or Merkle Tree, is a binary tree with labeled nodes. We represent the label for node n as $\Phi(n)$. If n is an internal node with children n_{left} and n_{right}, then

$$\Phi(n) = h(\Phi(n_{left})||\Phi(n_{right})) \quad (1)$$

Labels for leaf nodes are computed using data values depending upon the application, e.g., in case of a relation, a label is calculated as the hash of the tuple represented by that leaf.

3) Digital Signatures: A digital signature serves a role similar to regular signatures: i) it enables the recipient to ascertain without doubt the author of a message; ii) it prevents forgery – i.e., it is (computationally) infeasible for Alice to sign a message with Bob's signature, and iii) the signature is not reusable – i.e., a signed message cannot be used to generate a signature for another message.

TABLE I
SYMBOL TABLE

Symbol	Description
t_i	the ith tuple of a relation
$h(x)$	the value of a one way hash function over x
$\Phi(x)$	label of node x in MB-Tree or MHT
$a\|\|b$	concatenation of a and b
VO	a verification object
T_i	the ith committed transaction
DB_i	the consistent DB state after the ith commit
MBT_i	the proof structure after the ith commit
$Proof_i$	the MB-tree root label after the ith commit
$S_U(M)$	message M signed by U
C_h, C_{IO}	Cost of computing one hash and one disk IO, respectively
C'_{IO}	Cost of searching and retrieving one block from history
S_h, S_n, S_t	Size of a hash value, MB-tree node, and a tuple in the user table, respectively

B. Model

There are three main entities involved: **Alice**, the database *owner*; **Bob**, the *(untrusted) database server* that will host the database; and **Carol**, the client(s) that will access this data (may include Alice) from the server. Clients are authorized by Alice and can independently authenticate themselves with the server. In our model, multiple clients can run transactions concurrently. These transactions can modify the data as well. Figure 1 shows the various entities in this model.

Alice and Carol need to be ensured that the database is operated faithfully – i.e., all the transactions are executed correctly and the data are not maliciously modified by Bob or an attacker. Bob is interested in hosting Alice's database (possibly in return for a fee) and will thus make efforts to ensure that Alice is convinced about the fidelity of the hosting. Bob controls the hardware and software that is used to host the database. He has complete control over all the data. He has unconditional read and write access to the data, and can intercept all queries posed to the database and their results, and may even modify the stored data or the results sent back to Carol. Note that our assumptions about Bob are minimal. In most settings, the server is likely to be at least semi-honest – i.e., it will not maliciously alter the data or query results. However, due to poor implementation, failures, over commitment of resources, or other reasons, some loss of data or updates may occur. Given the lack of direct control over the server, Alice should not assume that Bob is infallible.

C. Correctness and Completeness

We begin by discussing the use of Merkle Hash Trees to prove correctness. And then further discuss a variant, the MB-tree, which we use as a building block for our overall solution.

1) Correctness: Correctness requires that all values that are part of a query result are indeed part of the database. In other words, this implies that the values returned are indeed from the right consistent state of the authentic database, and not *manufactured* by the server or an attacker.

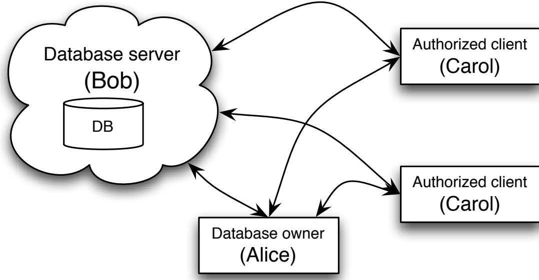


Fig. 1. The various entities involved: The database owner (Alice); The database server(Bob); and Authorized clients (Carol).

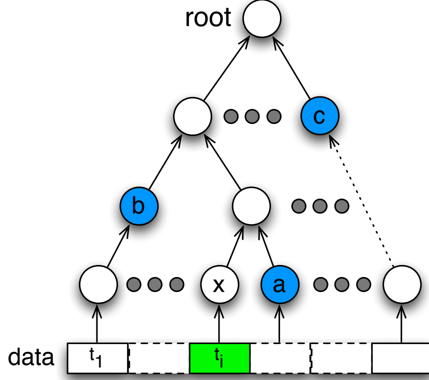


Fig. 2. An example Merkle tree

Merkle trees can be used to establish the correctness of query results from an outsourced database. Initially, when Alice chooses to outsource a database relation, she computes a Merkle tree over the relation. Alice saves only the value of the root label. The data are then sent to Bob for servicing queries. Bob computes the same Merkle tree structure over this data and then services queries from Alice. Figure 2 shows an example of the tree produced.

The correctness of a tuple, t_i, in the result of any query is established as follows: Alice requests the *Verification Object* (*VO*) for tuple t_i. Let x be the leaf node whose label was computed as the hash of t_i. The *VO* for t_i consists of all sibling nodes along the path from x to the root of the Merkle tree. In Figure 2, these nodes are shaded and are: a, b, c. Bob returns to Alice the *VO* for t_i. Alice uses the value of t_i to compute the label of x. She uses this label and the label for a from the *VO* to compute the label for x's parent, etc., up to the root label. If this value is the same as the root label that she had computed before outsourcing the data, she is convinced that t_i must be part of the original table. If not, then Alice is unconvinced that the database is uncompromised.

Note that the hash function has to be public. The label of the root saved by Alice is known as the *proof*. Of course, it is also possible for Carol to similarly submit queries and verify the correctness of each answer. All she needs to know is the value of the root label initially computed by Alice.

2) Completeness: Completeness requires that all values that should be in the answer to a query are indeed present in the answer. In other words, Bob has evaluated the query correctly and has returned all tuples that are produced as a result without dropping any out. Note that correctness and completeness together ensures the correctness of a read-only query.

As explained before, a Merkle Tree is a binary tree. However, it can be easily extended to use a B+-tree instead [1]. An MB-tree behaves just like a regular B+-tree with its nodes extended with child hash values. Similar to the Merkle Tree, the label of each non-leaf node is computed by hashing the concatenation of the labels of its children. The label of a leaf node is computed by hashing the concatenation of the hash values of each tuple entry in the node.

We use MB-trees to establish the completeness of the results of a query. Figure 3 shows an example tree for establishing completeness. Consider a range query which returns tuples $t_0 \ldots t_r$. To establish the completeness of tuples satisfying a range, the *VO* consists of the tuple just before the range (t_{-1}) and the tuple just after the range (t_{r+1}). *VO* then includes the path from those two tuples to the root.

To establish correctness, Alice computes the labels for all tuples in the results and the two surrounding values. She then computes the labels of ancestor nodes working her way up the tree. The *VO* contains the labels of nodes that she needs to compute the ancestor label and also which tuples belongs to the same leaf node. Finally, she compares the computed *proof* with the *proof* she stored earlier to determine if the result set contains all the tuples that should have been returned. If this is the case, she is assured that all valid tuples were returned to her. Alice verifies that t_{-1} and t_{r+1} are outside the query range, ensuring that the query results were indeed complete.

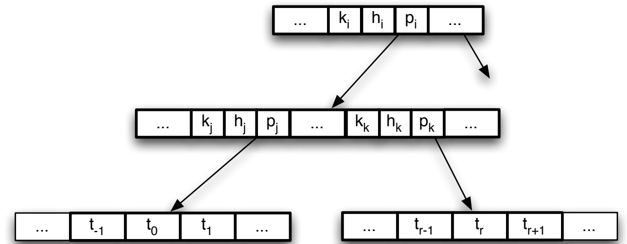


Fig. 3. An example MB tree for completeness. k_i represents a key, p_i and h_i represent pointer and hash value of the ith child. The bold values are returned as *VO* when the query result is the tuples $t_0 \ldots t_r$.

3) Updates Generated By Alice: If Alice wishes to update the data that she has previously outsourced to Bob, she can do so by simply sending the updates to Bob after recomputing the Merkle Tree structure incorporating the updates. Bob similarly applies the updates to the structure that he maintains. All subsequent queries are then answered using the updated database at Bob's server, and verifications are now done against the new *proof*.

This solution requires that (i) all updates have to be routed through the data owner (Alice) and (ii) if Alice wants to verify the update, she has to do it before executing further updates. In the next section, when we discuss a more general solution for updates that are not generated directly by Alice, we will remove both of these requirements.

III. Transactional Integrity

As seen in the previous section, solutions for ensuring correctness and completeness have been proposed in earlier work. In this section we present our solution for ensuring transactional integrity of a dynamic database where multiple clients can independently run transactions.

Transactional integrity requires that each transaction is run against a consistent database containing all, and only the changes of all previously committed transactions, in order of their commits. Furthermore, any updates generated by the correct execution of transaction are reflected in the updated database upon commitment. From the point of view of authentication, we impose the following restriction: Transactional integrity will only be guaranteed for *Replayable Transactions*. A replayable transaction is one which is deterministic – i.e., the outcome of the transaction (the outputs it generates, the values of its updates and its decision to commit or abort) is completely determined by the values that it reads from the database and its input parameters. This assumption is no different than one that is currently made by database systems in order to ensure the well known ACID properties of databases (e.g., the ability for the database to automatically restart a running transaction if it is deadlocked).

A. Requirements

To ensure the integrity of the database, each transaction must be authorized by Alice or Carol. Alice needs to be ensured that the transaction was faithfully executed by Bob without any tampering of the results. Furthermore, any subsequent queries should be answered against this resulting database following the transaction's commitment. The problem is difficult because we must ensure that Bob does not:

- drop a transaction, i.e., claims to execute it, but does not;
- add an invalid transaction not authorized by Carol;
- alter the order of execution of transactions; or
- alter the data read by or modified by a transaction.

In addition, we also require indemnity for Bob: that is, if he faithfully executes all transactions received from Alice or Carol, it is impossible for Alice or Carol to implicate him, i.e., he must be able to prove his innocence.

B. Key Idea

Databases are complex systems, but they are built to ensure that the following simple definition of consistency is satisfied[2]. Figure 4 shows this graphically. The initial state of the database (DB_0) is considered to be a consistent state. The correct (isolated, atomic) execution of a transaction(T_i) over a consistent state(DB_{i-1}) takes the database to a new consistent state(DB_i). Of course, this is only a conceptual notion – in reality multiple transactions execute concurrently. Thus, in practice, the database is in an inconsistent state represented by the partial execution of concurrent transactions. However, when a transaction is allowed to commit it is certain that its execution is equivalent to having run in isolation against the consistent state produced by the execution of all transactions that have committed earlier (in the order of commits).

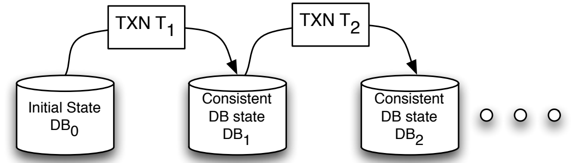


Fig. 4. A conceptual view of database consistency

Our initial solution focuses on strict serializability. If a user is willing to run a transaction with a weaker isolation level, than our solution can be applied with an appropriately relaxed notion of correctness. Since strict serializability is the most stringent condition for correctness and completeness (as required by many applications, e.g., banking), we limit our discussion to this case.

Our solution is built upon the fact that in strict serializable isolation level, each transaction "sees" a consistent, committed state of the database corresponding to the state produced after completing earlier committed transactions. All its reads are from this consistent state. If the transaction is able to successfully execute and commit, it generates a set of updates which must all be installed atomically to produce the next consistent state.

Based on this observation, from the conceptual point of view, the integrity and authenticity of a transaction's execution can be divided into three sub-components:

- Establishing that all values read by the transaction come from a single consistent state (in particular, one that reflects the updates of all prior committed transactions, in correct order).
- Faithful execution of the transaction using these values – determination of the correct values of the updates.
- Establishing that all updates generated by this execution have been applied to the database.

Under the assumption of replayable transactions, Bob does not need to prove the second point listed above (faithful execution) since this can be verified by a simple re-execution of the transaction over the same consistent state that was visible to the transaction when it was run by Bob. We need to establish that a given transaction, T_i, ran against a particular consistent state, DB_{i-1} and produced a particular consistent state, DB_i. This will be achieved by using the MB-tree structure discussed earlier. Although the database at any time is in flux and reflects an inconsistent state, we will only update the MB-tree structures at the time of transaction commitment. Thus each new MB-tree reflects the commitment of exactly one transaction. In other words, we maintain a one-to-one correspondence between the conceptual consistent states and the proof structure – a new version of the structure is generated in one step only upon the commitment of a transaction. This new structure is computed from the previous structure by applying the changes made by exactly one transaction – the one that has just committed. Bob has to declare this structure at the time of commitment of a transaction (by sending out beyond his control a signed copy of the new root label, $Proof_i$), and be able to use this structure when asked to

Algorithm 1 Initialization

1: Alice sends the initial database, DB_0 to Bob
2: Bob computes MBT_0, and sends $S_{Bob}(Proof_0)$ to Alice.
3: Alice independently computes $Proof_0$ and verifies what Bob has sent.
4: Alice retains $Proof_0$. (She may now choose to discard her copy of the database.)

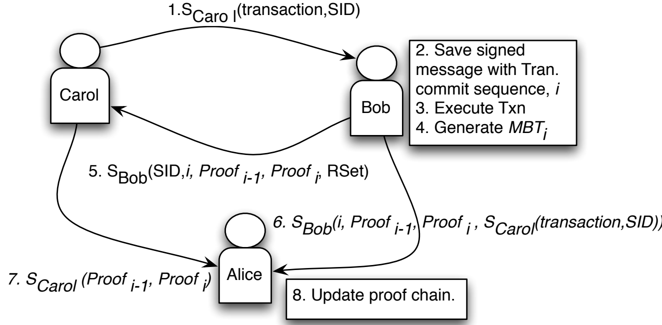


Fig. 5. Transaction Execution Steps

verify the correct execution of the transaction (which will also involve the structure before the commitment of the given transaction).

C. The Protocol

We now discuss our solution for ensuring Transactional Integrity. We discuss the initialization steps taken by Alice and Bob, the execution of a single transaction by Carol (similar for Alice), and the verification steps to establish the validity of a given transaction. For ease of exposition, we discuss only the case of one relation. The extension to multiple relations is straightforward and omitted due to lack of space. All signed messages are verified by the recipient – this is not explicitly mentioned in the discussion below.

While it is certainly feasible for Carol (or Alice) to validate every single transaction, this results in a heavy burden. In practice, we expect that they will randomly validate transactions with a frequency that reflects their distrust of Bob. In order for this to achieve the goals of this work, it is essential that they be able to verify *any* past transaction without any forewarning to Bob during the execution of the transaction. Moreover, the solution must ensure that once they request the validation of a transaction, it is impossible for Bob to go back and "fix" any errors or omissions with respect to the execution of that transaction. These requirements are met by our solution.

1) Initialization: Algorithm 1 describes the initialization step. In the beginning, Alice and Bob independently compute MBT_0 – the MB-tree structure over the initial state of the relation. Alice retains only the proof, $Proof_0$. Bob sets up the database with this initial table and opens shop, ready for processing transactions from Alice and Carol.

2) Transaction Execution: Algorithm 2 describes this step and Figure 5 shows the steps graphically. To execute a transaction, Carol sends to Bob a signed message containing the identity of the transaction (as discussed above), all necessary parameters (e.g., account numbers), and a unique transaction sequence number (SID). This sequence number must be unique for each transaction submitted by Carol (if there are multiple Carols, i.e., multiple authorized clients, the number needs to be unique for each such client, not necessarily across all clients). Bob verifies the signature to be that of Carol and examines the message. He rejects a transaction request from Carol if the sequence number is not larger than the earlier request from the same client. The sequence numbers prevent *replay* attacks by Bob. That is, Bob will be prevented from re-using a transaction request to run that transaction multiple times. The details will become clear later in the section.

Bob then runs the requested transaction. He keeps track of the data items written by the transaction. If the transaction successfully commits, Bob installs the updates produced by the transaction into the current proof structure. Since transactions are committed sequentially, let i be the ordinal position of this transaction's commitment since the initial outsourcing. This transaction will be identified by Bob as T_i – i.e., the ith transaction to commit. Concurrency control will ensure that this transaction's reads were consistent with DB_{i-1} – the consistent state corresponding to all earlier committed transactions. If this is not the case, then the transaction will not be allowed to commit (recall that we are assuming only strict, serializable executions[2]). Once it commits, its changes will be included in the next conceptual consistent state, DB_i. Bob stores the authorization message from Carol along with the transaction's commit sequence, i.

Bob needs to declare that T_i was applied on DB_{i-1} and produced DB_i. He does this by computing the corresponding MB-tree structures MBT_{i-1} and MBT_i. As part of the proof of the commitment of T_i, he send to Carol a signed message containing (i) the sequence number submitted by Carol, (ii) the transaction's commit sequence number, i, (iii) the label of the root of MBT_{i-1}, i.e., $Proof_{i-1}$, (iv) the label of the root of MBT_i, i.e., $Proof_i$, and (v) RSet, the result set produced by the transaction. He also sends to Alice a signed message containing (i) the transaction commit sequence number, i, (ii) $Proof_{i-1}$, (iii) $Proof_i$, and (iv) the Carol's signed request – $S_{Carol}(transaction, SID)$. Note that for a read-only transaction, Bob need not send anything to Alice.

Alice uses the messages from Bob to maintain the sequence of proofs: $Proof_0$, $Proof_1$, ..., $Proof_i$ that Bob claims to be the sequence of consistent states that the database has gone through. She ensures that $Proof_{i-1}$ is currently the last value in its proof chain. If this is the case, she adds $Proof_i$ to the end of the chain. She also retains the latest SID used by each client. She checks to see that the SID has not been used by this client earlier and is in increasing order for the client. Alice receives $Proof_{i-1}$ and $Proof_i$ from Carol as well, and ensures that $Proof_{i-1}$ precedes $Proof_i$ in the sequence received from Bob. If not, Alice has detected a problem.

3) Transaction Verification: Algorithm 3 explains the verification protocol formally. Following the execution of a transaction, Carol (or Alice) can arbitrarily decide if she wants to verify a given transaction (current or past). To verify transaction T_i, three requirements need to be established.

Algorithm 2 Transaction Execution

1: Carol generates the unique transaction sequence number SID and sends $S_{Carol}(transaction, SID)$ to Bob.
2: Bob records this message after verifying the signature and executes the transaction.
3: If the transaction successfully commits, Bob computes MBT_i and sends $S_{Bob}(SID, i, Proof_{i-1}, Proof_i, RSet)$ to Carol. i is the transaction's commit sequence number, and $RSet$ is the result set produced by the transaction.
4: Bob also sends to Alice $S_{Bob}(i, Proof_{i-1}, Proof_i, S_{Carol}(transaction, SID))$.
5: Alice verifies Bob's signature and then adds $Proof_i$ to its chain of proofs after verifying that $Proof_{i-1}$ is at the end of the current chain. Alice also checks that the SID for this client is in increasing order.
6: Carol sends $S_{Carol}(Proof_{i-1}, Proof_i)$ to Alice.
7: Alice checks that $Proof_{i-1}$ and $Proof_i$ are contiguous proofs in its chain.

Firstly, Bob has to show that all values read by the transaction were indeed from DB_{i-1}[1](Steps 2 and 3). To do this, he needs to produce the verification objects for the reads from MBT_{i-1}. For now, let us assume that Bob maintains a copy of the MB-tree for each consistent state. We will revisit this issue later and propose a more efficient solution. Carol uses the Correctness and Completeness mechanisms discussed earlier to verify the reads against $Proof_{i-1}$(Steps 4 and 5).

Secondly, Carol needs to know the correct values of all updates generated by the given (replayable) transaction when run on a database corresponding to DB_{i-1}. Given a replayable transaction and the values read by the transaction (as declared by Bob and validated in the previous step), we need to determine the values of its updates. Given the replayable transaction and the values that it reads, Carol (or Alice) can determine the values of its output and updates (Step 6).

Thirdly, Carol needs to establish that the updates of the transaction were faithfully recorded in the database and used for subsequent transactions. To do this, Bob needs to show that MBT_i differs from MBT_{i-1} by exactly the modifications of T_i. For this, Bob sends the node values from MBT_{i-1} which were modified at the time of commit of transaction T_i. Bob also sends other nodes values required for Carol to generate the $Proof_i$ (Step 7). Then, Carol can update the partial MBT_{i-1} to include the changes applied by the transaction and verify the new proof (Step 8). Carol then ensures that the new proof ($Proof_i$) is indeed what Bob claimed it to be at the time of transaction commit. This is verified by comparing it with $Proof_i$ value obtained from Alice in Step 4.

Notice that in our solutions, the overhead for Alice is minimal. Alice just maintains the proof chain and stores the latest SID used by each client. Alice incurs the cost of verification only if she wants to. The clients can verify the

[1]As is always the case, if T_i updates a data value and subsequently reads it, it will read the value it wrote, not the one from DB_{i-1}. This should be taken care of during the transaction verification.

Algorithm 3 Transaction Verification

1: Carol asks Bob to verify a transaction T_i
2: Bob computes MBT_{i-1} and MBT_i.
3: Bob sends to Carol the verification objects for all values read by T_i based on MBT_{i-1}.
4: Carol obtains $Proof_{i-1}$ and $Proof_i$ from Alice.
5: Carol verifies the correctness and completeness of T_i's reads.
6: Carol determines the outputs and updates for T_i (replays T_i) given these reads.
7: Bob sends to Carol the verification objects for T_i's updates based on MBT_i.
8: Carol verifies that MBT_i contains these updates.

transactions independently. There is a communication cost for Alice, as both Bob and Carol update Alice about the execution of a transaction. However, it does not stop Bob or Carol from executing further transactions. Hence, the communication can occur asynchronously.

D. Discussion of Correctness

We now show that the proposed protocol meets our requirements. We show how a failure on the part of Bob will be detected by our protocols.

Lemma 1: If Bob (or an intruder) maliciously modifies a set of tuples, $mSet$, after transaction T_i (with corresponding proof $Proof_i$), then $Proof_{i+1}$ will not authenticate the malicious version of mSet.

Proof: Any tuple value after executing T_i can be authenticated using $Proof_i$, which is declared by Bob after execution of T_i. If the values in the tuple set $mSet$ were modified after T_i, $Proof_i$ will not authenticate the updated values in $mSet$, i.e., the server will not be able to prove that the new values in $mSet$ were indeed part of the database after executing T_i. Thus, transaction T_{i+1} will not read those values (if it does, the server will not be able to authenticate those values). Since the execution of the transaction depends solely on the data that it reads, the verifier can generate the updates that the T_{i+1} generated. Thus the calculation of the new proof will not include the malicious changes to $mSet$. Hence, $Proof_{i+1}$ will not authenticate the changes in $mSet$. ■

Theorem 1: If Bob modifies $mSet$ after transaction T_i and if T_k is the first transaction after T_i that accesses the tuples in mSet, then verification of T_k will fail.

Proof: For the verification of T_k, the server has to authenticate the tuples that T_k reads against $Proof_{k-1}$. Using Lemma 1, we can say that T_{i+1} can not authenticate the malicious values of $mSet$. Applying the same lemma again on T_{i+1} ensures that T_{i+2} will not authenticate $mSet$ either, and so on. Hence, T_{k-1} will not authenticate the values in $mSet$ either. Hence, the verification will fail. ■

Theorem 1 establishes that each transaction reads data from a consistent state which reflects the updates applied by previously committed transactions. The proof chain stored at Alice establishes the order of commitment (serialization order) of the transactions. We now present different malicious events

that may compromise the trustworthiness of data, and discuss how our solution ensures that such events can be detected.

If Bob drops a transaction: Consider a transaction submitted by Carol that Bob pretends to execute (i.e., sends unauthentic responses to Carol, but does not actually execute the transaction). Bob has to notify Carol that it executed the transaction and her transaction modified the proof from $Proof_{i-1}$ to $Proof_i$ (for some i). As part of the protocol, Carol will send this information to Alice (Algorithm 2, Step 6). If Bob drops this transaction, Bob will claim that the next transaction (sent by the same client or some other client Carolina) moved the proof from $Proof_{i-1}$ to $Proof_i^{'}$. When Carolina sends this information to Alice, she will detect that Bob executed two transactions on the same consistent state, which exposes Bob's infidelity.

If Bob executes an unauthorized transaction: Bob can execute an unauthorized transaction in two ways: i) Bob could manufacture a new transaction and pretend that a client sent it to him; or ii) Bob could replay a transaction that it already executed. To manufacture a new transaction, Bob has to forge a client's signature as the protocol requires Bob to execute only signed transactions – this is computationally infeasible.

To prevent a replay of an old valid request, the protocol requires a unique, increasing identifier (SID) as part of the signed request. Hence an attempt to reuse an old signature will be caught when Alice receives $S_{Carol}(transaction, SID)$ value that does not show an increase in SID for the given client (Algorithm 2, Step 5).

If Bob does not run the transactions in the claimed sequence: The chain of proofs maintained by Alice prevents this from happening. Bob informs Carol of the commit order, i, for each transaction. The corresponding pair of proofs, $Proof_{i-1}$ and $Proof_i$ must validate this transaction. If Bob does not specify these correctly, verification of T_i will fail.

E. Indemnity for Bob

We also require that if Bob is honest and faithfully executes all transactions submitted by Alice and Carol, then he can prove his innocence. This is indeed the case for this solution.

We first consider Bob's indemnity from Carol. In order to verify a transaction submitted by Carol, Bob needs the following from Carol: (i) the request for running the transaction including the transaction name, its parameters, and a sequence number, and (ii) Carol's ability to replay a transaction faithfully. Carol cannot repudiate her request for running a transaction since she signs the request with all the necessary information. If Carol does not replay a transaction correctly, Bob can check that himself by replaying it and implicate Carol. This is possible because each transaction and parameters are known to each party, the values read from the database are known to Bob and he can verify that they are consistent with the consistent state corresponding to the proof value he sent to Carol in response to the transaction request. Thus, it is not possible for Carol to falsely implicate Bob.

Next we consider Bob's indemnity from Alice. Bob relies on Alice to maintain the chain of proofs and also to check that a given SID has not been used earlier for a given client. Alice cannot modify the chain with impunity. If she adds a proof that Bob has not provided, she would have to produce a signed message from Bob containing the old and new proofs. She cannot manufacture such a signed message. Similarly, she cannot delete any proof ($Proof_i$) from the chain, as she has to produce a signed message containing ($Proof_{i-1}$, $Proof_{i+1}$). If Alice makes a false claim that an SID value for a client is being reused by Bob, she has to manufacture a signed message from Bob containing the SID and client pair which is not computationally feasible.

Thus, Bob is protected from baseless claims of wrongdoing from either Alice or Carol, as desired.

F. History and Data Provenance

Recall that we assumed above that Bob maintains a copy of each successive MBT_i corresponding to the commit of each transaction. This is expensive and unnecessary. Instead, Bob can maintain a base structure and record incremental updates to the structure after each commit. Alternatively, he can maintain the latest version of the structure and maintain enough information to work backwards to an earlier version.

To reduce the storage cost, each tuple in the database and each node in MB-tree is assigned a unique id. Each tuple in the relation is also assigned a version number. A *history* stores each value that a tuple or a node takes as the database evolves. At the start, for each tuple and MB-tree node, the history stores only one value – the value in the database or MB-tree after the initialization step. Initially each tuple value is assigned version number 0. When a tuple or a node is modified, the version number is incremented by one and the new value is added to the history along with the transaction's sequence number which modified the value. As the database evolves, the history stores all the values that a particular tuple or MB-tree node takes at consistent states. When a client wants to verify an old transaction, the database server can use the history to generate the values that the server read when executing this transaction.

The history also provides a secure provenance of the data. When asked for the provenance of a tuple, the server responds with a set of provenance records(history). Each provenance record has three components: the id of the transaction that created that value; the version of the tuple; and the value of the tuple. Lemma 2 shows that the existence of any provenance record returned by the server can be verified. Further, Lemma 3 establishes that the completeness of the provenance records for a tuple (i.e., no record that should have been in the provenance is missing), can be verified. Theorem 2 then proves that the data provenance of a tuple returned by the server is indeed trustworthy. If not, the client will be able to detect the error.

Lemma 2: Any provenance record returned is indeed an authentic record.

Proof: A provenance record includes the transaction that modified the tuple and the new tuple value. The authenticity of a provenance record can be ensured by verifying the transaction that generated that tuple value. ■

Lemma 3: Any provenance record that should have been returned is indeed in the provenance.

Proof: Since each tuple value is attached with its version, when the server returns the provenance of a tuple, $< T_{i1}, 0, v_0 >, < T_{i2}, 1, v_1 > \ldots < T_{ij}, j, v_j >$, the version numbers have to be contiguous, starting with 0. Any missing version number will mean incompleteness. To ensure that v_j is indeed the last version of the tuple, the client asks the server to verify the authenticity of v_j against the latest proof. ■

Theorem 2: The provenance of a tuple given by the server is correct and complete. If not, the client can detect the error.

Proof: The theorem is a direct consequence of Lemma 2 and Lemma 3. ■

G. Efficiency

For ease of exposition, the protocol discussed above intentionally omits several possibilities for gaining efficiency. We now discuss some of the possible optimizations.

As discussed before, in our solutions, the overhead for Alice is minimal. Alice just maintains the proof chain and stores the latest SID used by each client. In practice, the size of the proof chain will be small compared to the database size. Alice incurs the cost of verification only if she wants to. The clients can verify the transactions independently. There is a communication cost for Alice, as both Bob and Carol update Alice upon the commitment of a transaction. However, the proposed solution does not prevent Bob or Carol from executing further transactions before sending updates to Alice. Hence, transaction processing need not stall while updates are sent. Updates can be delayed as long as they arrive before verification is performed. Similarly, verification can be performed at any later time, and is not limited to immediately after transaction execution.

In the above solution, Bob has to maintain *history*, which he uses to generate the values that the server read when executing an old transaction. The size of this information can grow to be very large. If space is a problem, we can introduce *verification checkpoints*. A verification checkpoint corresponds to a statute of limitations for Alice – i.e., we do not allow Carol or Alice to verify any transaction beyond a certain time in the past (e.g., a month). After this point, if the transaction is not challenged, Bob can assume that Alice accepts it and he can discard any data necessary to verify that transaction.

H. Analysis

We now analyze some of the overheads introduced by our protocol. We provide a treatment similar to that in [1] which introduced the MB-tree and serves as a base case for us since it provides a solution that meets the requirements of correctness and completeness, and also allows for centralized updates through Alice. It does not provide a solution that meets the Transactional Integrity requirement. In our solution, the client's cost of verifying an update or read is the same as that with an MB-tree, as clients still verify an update or query against an MB-tree.

Authentication Structure Construction Time: For a database with n tuples and fanout f, the cost of construction of our data structures involves calculating hashes for each tuple and each node in the tree. Also, it requires the cost of writing those nodes to disk. For a tree of height d, the number of nodes in the tree will be:

$$m = \frac{f^d - 1}{f - 1} = O(n) \tag{2}$$

Hence, the cost of construction is:

$$nC_h + 2mC_h + 2mS_nC_{IO} + nS_tC_{IO} \tag{3}$$

where S_n and S_t are the sizes of a tree node and a tuple, respectively. C_h and C_{IO} are the costs of computing a hash value, and one block IO, respectively. Thus, like an MB-tree, the construction time is $O(n)$ – linear in the size of the database. This cost exceeds that of an MB-tree by $mS_nC_{IO} + nS_tC_{IO}$, due to the overhead of the history.

Update Time: To insert or delete a tuple from the tree, the path from the leaf node to the root has to be updated. Also, the updated values have to be added to the history. Hence, the cost of an update is:

$$C_h + dC_h + 2dS_nC_{IO} \tag{4}$$

This is $O(\log n)$.

VO Construction Cost: To construct a VO, the server has to go through the history and find the values that it read while executing a transaction. If $C^{'}_{IO}$ is the cost of finding the value of a tuple or node which the transaction read, then the cost of VO construction is:

$$2dC^{'}_{IO} \tag{5}$$

Since the height of the tree is d, the server has to find the rightmost path and the left most path to construct VO. $C^{'}_{IO}$ will increase as the transaction count increases, since it increases the history size hence search space.

Storage Overhead: Since our proposed data structure also stores the history of each tuple and tree node, the server keeps an extra copy of the tree and the relation. Thus, in the start we need twice as much disk space as that required by an MB-Tree. For each update, the server keeps a copy of the updated data in history. Thus, after k updates, the storage cost is:

$$2nS_t + 2mS_n + kdS_n + kS_t \tag{6}$$

Thus the overhead for Bob is $O(n + klog(n))$ – i.e., linear in the size of database and updates. On the other hand, Alice has to store the proof chain (one hash and one transaction ID per transaction) and the largest SID values for each client. This requires a disk space of:

$$(|h| + |tID| + |SID| + |User|)t \tag{7}$$

Thus the overhead for Alice is minimal and is linear in the number of transactions and clients.

In the section next, we discuss implementation details and an empirical evaluation of the proposed solutions.

IV. Experiments

To demonstrate the feasibility and evaluate the efficiency of the proposed solution, we implement our protocols with the MB-Tree in Oracle. Our implementation is built on top of

Oracle without making any modifications to the internals. *Note that even though Oracle uses "read committed" isolation by default, it allows strict serializability. All our experiments are run with strict serializability.* The protocols are implemented in the form of database procedures using Pl/SQL. While we expect that the ability to modify the database internals or to exploit the index system will lead to a much more efficient implementation, our current goal is to establish the feasibility of our approach and to demonstrate the ease with which our solution can be adopted for any generic DBMS. Clients are implemented using Python.

Setup: We create a synthetic database with one table *uTable* containing one million tuples of application data. *uTable* is composed of a table with two attributes ($TupleID$ and A). The table is populated with synthetic data taking random values of A between -10^7 and 10^7. All necessary structures for the protocol are maintained using other tables. Table II describes the different tables and indexes used in our prototype. An MB-tree is created on attribute A (integer). We consider three transactions implemented as stored procedures, namely *Insert*, *Delete*, and *Select*. *Insert* creates a new tuple with a given value of attribute A. *Delete* deletes the tuples which have the given value of attribute *A* and *Select* is a range query over attribute A. In practice, transactions will be more complex than a single insert or delete. Our solution can handle complex transactions as well. However, for simplicity, we consider only simple transactions. The experiments were run on an Intel Xeon 2.4GHz machine with 12GB RAM and a 7200RPM disk with a transfer rate of 3Gb/s, running Oracle 11g on Linux. We run Oracle with a **strict serializable** isolation level. We use a standard block size of 8KB.

Implementation Details: We implement the MB-tree in the form of a database table. Each tuple in the MB-tree table represents a node in the tree. A better way to maintain the MB-Tree would be to use the B+ index trees of the database. However, that will require internal modifications to the index system of the database. We leave that for future work. Table *uTableMBT* stores the MB-tree for the data in *uTable*. Each MB-tree node is identified by a unique id. Each node stores keys in the range $[key_min, key_max)$. $level$ denotes the height of the node from the leaf level, i.e., leaf nodes have $level = 0$, and the root has the highest level. The $keys$ field stores the keys of the node, and the $children$ field stores the corresponding child ids and labels. Finally, $Label$ stores the label of the node. This table is updated at the time of transaction commit.

Tables *uTableHistory* and *uTableMBTHistory* are used to store the history of the tables *uTable* and *uTableMBT* respectively. When a tuple is modified in *uTable* or *uTableMBT*, a new tuple is inserted in the corresponding history table to store current values. For example, when a new tuple is created in *uTable* by a transaction with transaction ID *tID*, an entry is added to *uTableHistory* with the value of the new tuple and transactionID as *tID*. At the time of a commit, the transaction modifies the MB-tree to update the proof. The updated node values are inserted into *uTableMBTHistory* with transactionID *tID*. Updates to the MB-tree are made level by level, beginning at the leaves and working to the root. Once the root is updated, the transaction is committed.

A. *Results*

We now present the results of our experiments. To provide a base case for comparison, we compare the performance of our protocol with a regular MB-tree based protocol [1] where all updates are routed through the data owner. Furthermore, this solution does not provide indemnity for the server or secure provenance. We analyze the costs of construction for the authentication data structures, execution of a transaction, and verification of a transaction. We also study how our solution scales with multiple clients concurrently running transactions.

The fanout for the authentication structure is chosen so as to ensure that each tree node is contained within a single disk block. In each experiment, time is measured in seconds, IO is measured as the number of blocks read or written as reported by Oracle, and storage usage is measured in number of file blocks. The reported times and IO are the total time and IO for the entire workload. Each experiment was executed 3 times to reduce the error – average values are reported. In the plots, MBT represents the protocol from [1] where updates are always routed through Alice, and MBT^* represents our protocol where updates are sent directly from the clients to the server. In experiments that measure the effect of multiple clients concurrently running transactions, we keep the total number of transactions constant. We divide the workload equally among multiple clients. Figure 6 shows the common legend for all the result graphs.

Construction Cost: First, we consider the overhead of constructing (bulk loading) the proposed data structure. For our solution, there is an extra cost for storing the history of the database, which increases the storage cost and construction time. Figures 7(a) and 7(b) show the effect of data size on construction time and storage overhead, respectively. As predicted by our analysis (Eqns. 3 and 6), both costs increase linearly with the size of the database. In addition to the MB-tree, we maintain an extra copy of the MB-tree and the database table in the history files. Hence, our solution incurs a 100% storage overhead. The construction time has two components; time to compute hashes and time for IO (Eqn. 3). Our protocol needs twice the amount of IO as compared to the MB-tree. However, the IO cost is superseded by hash computation. Hence, the construction time is not much higher than maintaining just an MB-tree.

In past work, the verification of a transaction was only allowed immediately after its execution, i.e., before any other transactions are executed. *Our work removes this restriction and enables the verification of past transactions. This provides much greater flexibility and reduces the need to immediately verify transactions.* Of course, the added functionality comes at an additional storage cost for the history tables.

Insert Cost: We now discuss the cost of inserting and deleting tuples. Since both operations show similar costs, we only present the results for insertion due to lack of space.

TABLE II
RELATIONS AND INDEXES IN THE DATABASE

Table	Attributes	Indexes
uTable	TupleID, A, Version#	A
uTableHistory	TransactionID, TupleID, A, Version#	(TupleID, TransactionID)
uTableMBT	id, level, Label, keys, children, key_min, key_max	id, (key_min, key_max, level)
uTableMBTHistory	TransactionID, id, level, Label, keys, children, key_min, key_max	(TransactionID, id)
Transaction	id, query, finalLabel	id

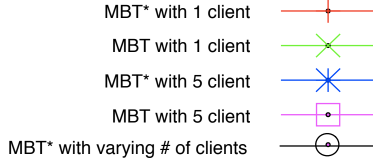
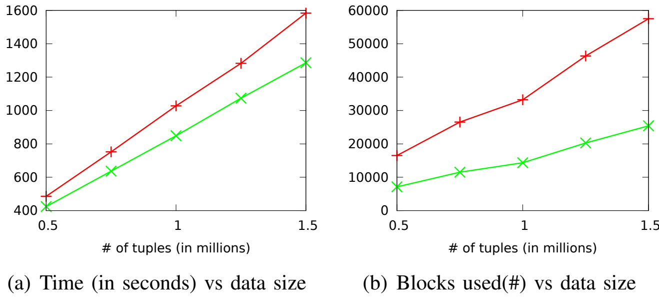
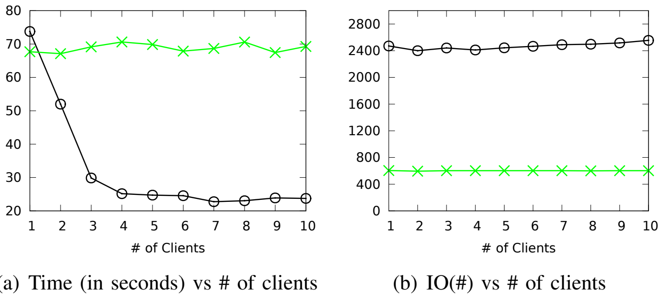


Fig. 6. Legend used for all result graphs.



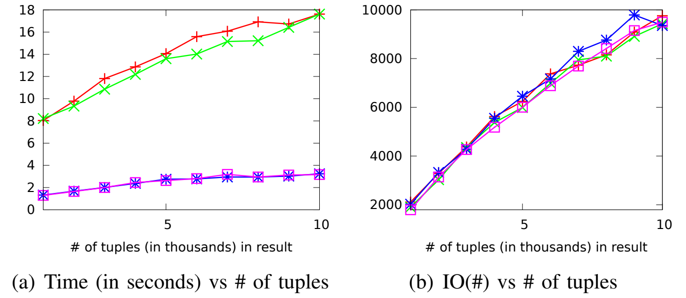
(a) Time (in seconds) vs data size (b) Blocks used(#) vs data size

Fig. 7. Construction time and storage overhead



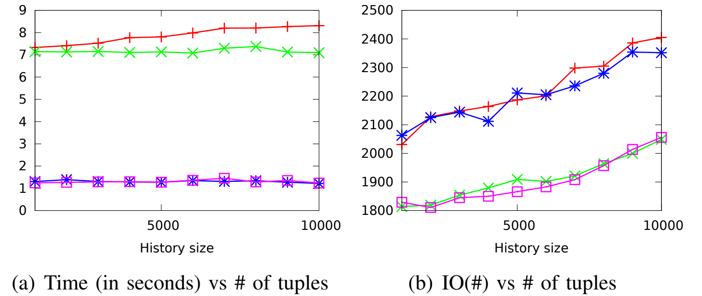
(a) Time (in seconds) vs # of clients (b) IO(#) vs # of clients

Fig. 9. Cost of Insert vs number of clients



(a) Time (in seconds) vs # of tuples (b) IO(#) vs # of tuples

Fig. 11. Cost of search+verification vs number of tuples in the result



(a) Time (in seconds) vs # of tuples (b) IO(#) vs # of tuples

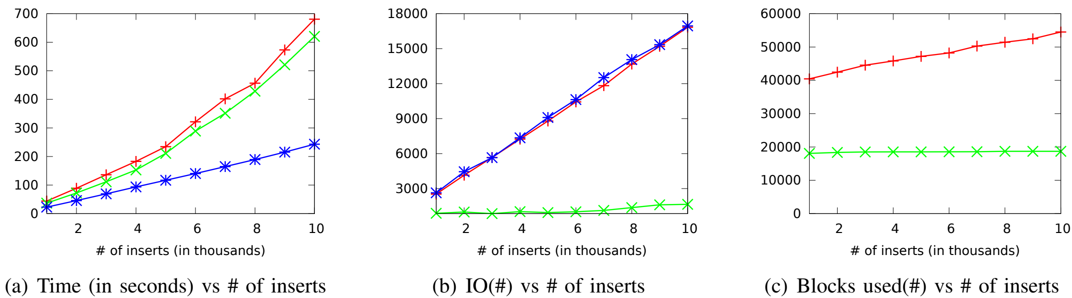
Fig. 12. Cost of search+verification vs History size (number of inserts before doing search)

For this experiment, no verification is performed. In the first experiment, we study the performance as the number of *Insert* transactions is increased. Figures 8(a), 8(b) and 8(c) show the results. As expected, with a single client, our protocol incurs a much higher overhead for storage and IO for maintaining the history information. These costs increase linearly with the number of transactions (Eqn. 6) Surprisingly, this does not translate into a significant increase in the running time. This represents the computational overhead of hashing and concatenations which dominate the cost (see Eqn. 4).

A key advantage of our protocol comes to light as we begin to increase the number of concurrent clients, as seen in Figure 8(a) where the running time for our protocol drops significantly when 5 clients run the same number of transactions in total, i.e., each client runs one fifth of the total number of transactions. In order to better study the impact of concurrent clients, we ran another set of experiments where a varying number of clients ran a total of 1000 *Insert* transactions. The results are shown in Figures 9(a) and 9(b). As we can see, the MB-tree solution which needs to process all updates through a single node (Alice) sees no gain in performance, whereas our solution results in improved performance with greater concurrency (even though it is performing a much larger amount of IO).

Verification Cost: We now demonstrate the overhead of transaction verification on the system. We ran 1000 *Insert* transactions with increasing fractions of transactions that are verified. The percentage of transactions that are verified reflects the data owner's distrust of the server. Figure 10(a) and 10(b) show the results. As the verification percentage increases, we observe that the execution time of the transactions increases. However, disk IO does not increase as rapidly since there is little extra IO for verification as compared to running the transaction itself. Verification also takes advantage of already cached MB-tree nodes.

Figure 10(c) shows the effect of the increase in number of clients on execution time. For this experiment, we ran 1000 *Insert* transactions and each transaction was verified. As before, our prototype scales much better to the increase in the number of clients. This is because verification can be

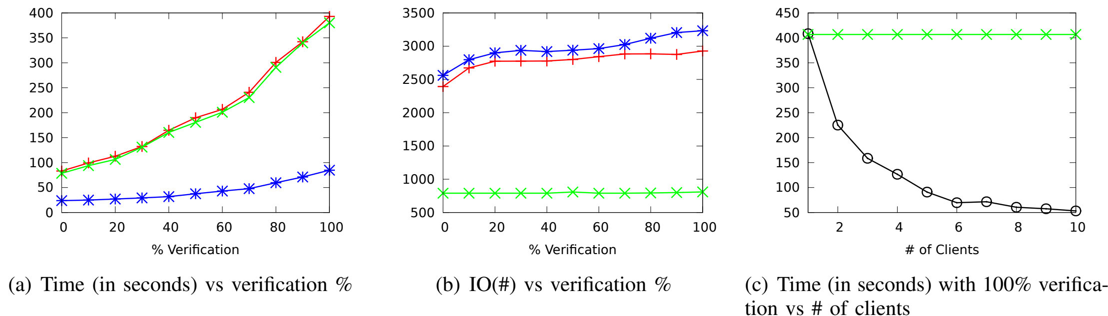


(a) Time (in seconds) vs # of inserts

(b) IO(#) vs # of inserts

(c) Blocks used(#) vs # of inserts

Fig. 8. Cost of Insert



(a) Time (in seconds) vs verification %

(b) IO(#) vs verification %

(c) Time (in seconds) with 100% verification vs # of clients

Fig. 10. Cost of Insert and verification

done independent of transaction execution. Hence, verification executes while other transactions are running. Whereas for the case of just an MB-tree, the verification of a transaction prevents other transactions from running.

Search Cost: Now we evaluate the performance of our solution for range queries (*Search*). This cost is influenced by both the size of the result (larger results will be more expensive to verify) and the size of the history that needs to be searched for generating the proof. We conduct two experiments to study this behavior. In the first experiment, we run 1000 *Insert* transactions on the database to populate history. Then, we run 100 *Search* transactions with 100% verification for different ranges (thereby with different result set size). Figures 11(a) and 11(b) show the results. As the result set size increases, execution time and the amount of IO increase. For verification, the server has to return the right and left most paths of the range. Along with this, the server also has to return which tuples belong to which leaf nodes, as that is crucial information for the client to be able to verify the result set. Hence, as the result set size increases, the verification object size increases which results in an increase in verification time. The performance of our solution is comparable to that of an MB-tree alone.

In the second experiment, we vary the history size by executing varying numbers of *Insert* transactions before running a fixed search and verification. Figures 12(a) and 12(b) show the effect of increase in history size. The x-axis in both graphs represents the number of *Insert* transactions executed before running the search transactions. As expected, increase in the history size increases the verification time for our solution. However, it does not increase rapidly. Note that the search time without verification for our solution and MB-tree was the same, hence we do not report it separately. Overall, we see that the proposed ideas can be easily implemented on top of an existing DBMS. Even with this simple implementation, the overhead for ensuring transactional integrity is reasonable and actually less than the cost of the state of the art for ensuring only correctness and completeness. As stated earlier, we believe there is a lot of room for more efficient implementations. We will explore optimizations in future work.

V. Related Work

The problem of ensuring the authenticity of query results from an untrusted (e.g., outsourced) database has been explored by several researchers [3], [1], [4], [5], [6], [7]. Some of the earlier work only considered correctness of results [3], [8], while later work consider both correctness and completeness [1], [5]. A few of these [1], [5] have considered updates. In most of these works, it is assumed that a single, central entity executes the updates. This is an unreasonable assumption for many applications. Only limited work has been done for the situation where multiple clients can update the data [5]. To the best of our knowledge, no work has been done towards transactional integrity with multiple clients.

[1] offers an example of a single updater solution. It proposes an embedded Merkle tree (EMB tree) for query correctness and completeness. An EMB tree is an embedded B+-tree similar to Merkle Hash tree. The root hash of the tree is made available to clients. With the help of this root hash, clients can prove the correctness and completeness of their

query results. Updates are performed only by the data owner and the updated root label is then distributed to the clients.

[5] proposes a solution for the multi-owner model. It allows updates from multiple clients using the BGLS [9] signature scheme, which makes it easy to handle signatures from multiple clients. Signatures from different clients can be aggregated together, and can be verified in a single step. For proving completeness, signature chains are used, where each tuple is signed together with the tuple just before and after it in sorted order. This work does not handle transactional semantics. Furthermore, the approach has been shown to be orders of magnitude slower than hashing based schemes [1].

Much work has been done towards data provenance and tamper-proofing of data [10], [11], [12], [13], [14]. While most works focus on storing and querying provenance [10], [13], some have considered the problems of privacy and trustworthiness of data provenance [11], [12].

Researchers have also begun to consider similar problems with respect to generic data. For example, [15] presents a solution to ensure that large files that have been outsourced are indeed available for download in their entirety. The solution provides a means to ensure that if parts of such files become corrupted or missing, then the client is able to discover this with high probability. [16] presents a solution to the problem of ensuring correct execution of a CVS server. In this work, it has been proved that to prove the execution of CVS (which includes both read and write), users have to communicate with each other. The need for communication arises to ensure the *freshness* of the data. [17] uses the notion of fork-consistency to ensure integrity of revision control systems.

Some work has also been done towards ensuring the integrity of databases using trusted hardware [18], [19]. In these systems, trusted hardware (e.g., a secure co-processor) is used to execute queries correctly and with privacy. Other orthogonal avenues of research focus include privacy preservation in outsourced databases [20], [21], [22], [23], and intrusion detection [24], which studied the problem of detecting malicious modifications of data by an external intruder. This is achieved through tamper detection of an audit log of the database that records all changes. This work assumes that the database owner is a trusted entity. Our solutions (with slight modifications) can be applied to solve the same problem.

VI. Conclusion

In this paper we introduced the problem of ensuring the authenticity and integrity of dynamic transactional database hosted on an untrusted server where the data owner may not have any direct control over the database. To the best of our knowledge, this problem has not been identified in earlier work. We develop novel solutions for this problem. Our protocol makes it possible to detect any failures on the part of the server to faithfully host a transactional database with multiple independent clients. Furthermore, the solutions also provide indemnity for the outsourcing server against false claims of erroneous processing, and provide assured provenance for the data managed by the untrusted server. These solutions are the first to address the problem of transactional integrity of an untrusted database. We demonstrate that the solutions can be implemented over an existing database system (Oracle) without making any changes to the internals of the DBMS. Our results show that we are able to remove the need to trust the server and provide support for independent clients at a cost comparable to earlier work that does not provide either of these guarantees. We believe that the efficiency of the solutions can be further improved by modifying the internals and also developing proof structures that have better disk performance (e.g., using GiST like indexes). We plan to explore these issues in future work.

VII. Acknowledgements

The work in this paper is supported by National Science Foundation grants IIS-1017990 and IIS-09168724.

References

[1] F. Li, M. Hadjileftheriou, G. Kollios, and L. Reyzin, "Dynamic authenticated index structures for outsourced databases," in *SIGMOD*, 2006.
[2] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*. Addison Wesley, 1987.
[3] P. T. Devanbu, M. Gertz, C. Martel, and S. G. Stubblebine, "Authentic third-party data publication," in *DBSec*, 2000.
[4] E. Mykletun, M. Narasimha, and G. Tsudik, "Authentication and integrity in outsourced databases," in *NDSS*, 2004.
[5] M. Narasimha and G. Tsudik, "Authentication of outsourced databases using signature aggregation and chaining," in *DASFAA*, 2006.
[6] H. Pang, A. Jain, K. Ramamritham, and K. lee Tan, "Verifying completeness of relational query results in data publishing," in *SIGMOD*, 2005.
[7] S. Singh and S. Prabhakar, "Ensuring correctness over untrusted private database," in *EDBT*, 2008.
[8] H. Pang and K. Tan, "Authenticating query results in edge computing," in *ICDE*, 2004.
[9] D. Boneh, C. Gentry, B. Lynn, and H. Shacham, "Aggregate and verifiably encrypted signatures from bilinear maps," in *EUROCRYPT*, 2003.
[10] A. P. Chapman, H. V. Jagadish, and P. Ramanan, "Efficient provenance storage," in *SIGMOD*, 2008.
[11] S. B. Davidson, S. Khanna, S. Roy, J. Stoyanovich, V. Tannen, and Y. Chen, "On provenance and privacy," in *ICDT*, 2011.
[12] R. Hasan, R. Sion, and M. Winslett, "Introducing secure provenance: Problems and challenges," in *StorageSS*, 2007.
[13] G. Karvounarakis, Z. G. Ives, and V. Tannen, "Querying data provenance," in *SIGMOD*, 2010.
[14] J. Zhang, A. Chapman, and K. Lefevre, "Do you know where your data's been? — tamper-evident database provenance," in *SDM*, 2009.
[15] A. Juels and B. S. K. Jr., "Pors: Proofs of retrievability for large files," in *CCS*, 2007.
[16] M. Venkitasubramaniam, A. Machanavajjhala, D. J. Martin, and J. Gehrke, "Trusted cvs," in *STD3S*, 2006.
[17] C. Cachin and M. Geisler, "Integrity protection for revision control," in *ACNS*, 2009.
[18] S. Bajaj and R. Sion, "TrustedDB: a trusted hardware based database with privacy and data confidentiality," in *SIGMOD*, 2011.
[19] E. Mykletun and G. Tsudik, "Incorporating a secure coprocessor in the database-as-a-service model," in *IWIA*, 2005.
[20] R. Agrawal, J. Kiernan, R. Srikant, and Y. Xu, "Order preserving encryption for numeric data," in *SIGMOD*, 2004.
[21] B. Hore, S. Mehrotra, and G. Tsudik, "A privacy-preserving index for range queries," in *VLDB*, 2004.
[22] F. Kerschbaum and J. Vayssiere, "Privacy-preserving data analytics as an outsourced service," in *CCS*, 2008.
[23] H. Hacigümüş, B. Iyer, C. Li, and S. Mehrotra, "Executing sql over encrypted data in the database-service-provider model," in *SIGMOD*, 2002.
[24] R. S. Shilong, R. T. Snodgrass, S. S. Yao, and C. Collberg, "Tamper detection in audit logs," in *VLDB*, 2004.